Time-stamp: <2002-03-18 11:56:24 piet>

Pure Gravity

or

Particles at Play

Volume 1:

Writing an N-Body Code

Piet Hut
Institute for Advanced Study
1 Einstein Drive
Princeton, NJ 08540
U.S.A.
piet@ias.edu

Jun Makino
University of Tokyo, Dept. of Astronomy
7-3-1 Hongo, Bunkyo-ku
Tokyo 113-0033
JAPAN
makino@astron.s.u-tokyo.ac.jp

Chapter 8

A More Modular N-Body Hermite Code

8.1 Starting a Tool Box

In this chapter we will discuss in detail a more modular version of the Hermite code `hermite6.C`, developed in the previous chapter. The new version is called `nbody_sh1.C`. Here 'sh' stands for the shared but variable time step choice, and the number 1 indicates again that this is the first version. This code will be the first tool of a tool box that we will continue to develop in the rest of this book, as well as in following books in this series. From now on, each tool will adhere to our N-body I/O format, specified in the previous chapter (and possibly more fancy formats as well, but we will keep those more advanced versions compatible with our current bare bones format). In addition, each tool will have extensive comments, explaining both the usage and the internal structure of the code.

The new code, `nbody_sh1.C`, has roughly four times more lines than the previous version, `hermite6.C`. Almost half of these lines are either comments of blank lines, both of which help to make the code more readable and more understandable. The fact that the code itself still has more than twice the length of the previous version stems from several factors. First, the new code has nine functions, besides `main()`, while the old code had only two. Second, there are seven command line options, rather than two. Third, we now declare all functions at the top of the file. Finally, there is more diagnostics output than we had before.

Below, the full code is presented, one function at a time.

8.2 Gravitylab

Our aim is to build a powerful software environment for experiments in stellar dynamics of dense stellar systems. The idea is to build a virtual laboratory, which we will call *gravitylab*. From now on, each new tool in our tool box will have a distinctive 'gravitylab' header:

```
              // Time-stamp: <2002-01-18 21:51:36 piet>
             //================================================================
            //                                                                |
           //           /__----__                         ........            |
          //       .           \                     ....:        :.          |
         //       :                 _\|/_         ..:                         |
        //       :                   /|\         :                     _\|/_  |
       //  ___   ___                  _____                      ___    /|\   |
      //  /     |   \    /\ \     / |   |   \  / |        /\    |   \         |
     //  |   __ |___/   /  \ \   /  |   |    \/  |       /  \   |___/         |
    //   |    | |  \   /____\ \ /   |   |    /   |      /____\  |   \     \/  |
   //     \___| |   \ /      \ V    |   |   /    |____ /      \ |___/     |   |
  //                                                                      /   |
 //              :                       _/|     :..                    |/    |
//                :..               ____/           :....          ..         |
/*   o   //          :.    _\|/_     /                   :........:           |
 *  0  `//\                 /|\                                               |
 *  |     /\                                                                  |
 *=============================================================================
```

We will not show these headers in future code listings, but they will be there in the source code for other tools. The three stars moving on a figure-8 orbit are inspired by the solution presented in chapter 5. They are being observed at bottom left by the small figure looking through a telescope.

Note the time stamp at the very first line. This is a handy feature of the *emacs* editor that we have used to write this book. When you add the line "`(add-hook 'write-file-hooks 'time-stamp)`" to the `.emacs` startup file, the date and time and user name will be updated automatically each time you write the file to disk.

8.3 Introductory Comments

Immediately following the gravitylab header, we see a lengthy comment block:

```
 *=============================================================================
 *
 *  nbody_sh1.C:  an N-body integrator with a shared but variable time step
```

```
*               (the same for all particles but changing in time), using
*               the Hermite integration scheme.
*
*               ref.: Hut, P., Makino, J. & McMillan, S., 1995,
*                     Astrophysical Journal Letters 443, L93-L96.
*
*  note: in this first version, all functions are included in one file,
*        without any use of a special library or header files.
*-----------------------------------------------------------------------------
*
*  usage: nbody_sh1 [-h (for help)] [-d step_size_control_parameter]
*                   [-e diagnostics_interval] [-o output_interval]
*                   [-t total_duration] [-i (start output at t = 0)]
*                   [-x (extra debugging diagnostics)]
*
*         "step_size_control_parameter" is a coefficient determining the
*            the size of the shared but variable time step for all particles
*
*         "diagnostics_interval" is the time between output of diagnostics,
*            in the form of kinetic, potential, and total energy; with the
*            -x option, a dump of the internal particle data is made as well
*
*         "output_interval" is the time between successive snapshot outputs
*
*         "total_duration" is the integration time, until the program stops
*
*         Input and output are written from the standard i/o streams.  Since
*         all options have sensible defaults, the simplest way to run the code
*         is by only specifying the i/o files for the N-body snapshots:
*
*            nbody_sh1 < data.in > data.out
*
*         The diagnostics information will then appear on the screen.
*         To capture the diagnostics information in a file, capture the
*         standard error stream as follows:
*
*            (nbody_sh1 < data.in > data.out) >& data.err
*
*  Note: if any of the times specified in the -e, -o, or -t options are not an
*        an integer multiple of "step", output will occur slightly later than
*        predicted, after a full time step has been taken.  And even if they
*        are integer multiples, round-off error may induce one extra step.
*-----------------------------------------------------------------------------
*
*  External data format:
*
*     The program expects input of a single snapshot of an N-body system,
```

```
 *     in the following format: the number of particles in the snapshot n;
 *     the time t; mass mi, position ri and velocity vi for each particle i,
 *     with position and velocity given through their three Cartesian
 *     coordinates, divided over separate lines as follows:
 *
 *                      n
 *                      t
 *                      m1 r1_x r1_y r1_z v1_x v1_y v1_z
 *                      m2 r2_x r2_y r2_z v2_x v2_y v2_z
 *                      ...
 *                      mn rn_x rn_y rn_z vn_x vn_y vn_z
 *
 *     Output of each snapshot is written according to the same format.
 *
 *  Internal data format:
 *
 *     The data for an N-body system is stored internally as a 1-dimensional
 *     array for the masses, and 2-dimensional arrays for the positions,
 *     velocities, accelerations and jerks of all particles.
 *-----------------------------------------------------------------------------
 *
 *    version 1:  Jan 2002   Piet Hut, Jun Makino
 *-----------------------------------------------------------------------------
```

It starts with the name of the file, a brief summary with a reference to the literature, followed by a detailed description of how to use the code. For a typical user, this is all the information needed. As long as the user combines `nbody_sh1` with other tools from gravitylab, there is even no need to understand the external data format, in which the N-body snapshots are written to and read from files. For those users interested in such details, as well as in the internal format in which the data are stored during the execution of the code, the comment block contains format information near the end. The last few lines list the history and version numbers of the code.

8.4 Include Statements, Function Declarations, etc.

The first lines of real code start right after the introductory comments:

```
#include  <iostream>
#include  <cmath>                          // to include sqrt(), etc.
#include  <cstdlib>                        // for atoi() and atof()
#include  <unistd.h>                       // for getopt()
using namespace std;

typedef double  real;                      // "real" as a general name for the
```

```
                                          // standard floating-point data type

const int NDIM = 3;                       // number of spatial dimensions

void correct_step(real pos[][NDIM], real vel[][NDIM],
                  const real acc[][NDIM], const real jerk[][NDIM],
                  const real old_pos[][NDIM], const real old_vel[][NDIM],
                  const real old_acc[][NDIM], const real old_jerk[][NDIM],
                  int n, real dt);
void evolve(const real mass[], real pos[][NDIM], real vel[][NDIM],
            int n, real & t, real dt_param, real dt_dia, real dt_out,
            real dt_tot, bool init_out, bool x_flag);
void evolve_step(const real mass[], real pos[][NDIM], real vel[][NDIM],
                 real acc[][NDIM], real jerk[][NDIM], int n, real & t,
                 real dt, real & epot, real & coll_time);
void get_acc_jerk_pot_coll(const real mass[], const real pos[][NDIM],
                           const real vel[][NDIM], real acc[][NDIM],
                           real jerk[][NDIM], int n, real & epot,
                           real & coll_time);
void get_snapshot(real mass[], real pos[][NDIM], real vel[][NDIM], int n);
void predict_step(real pos[][NDIM], real vel[][NDIM],
                  const real acc[][NDIM], const real jerk[][NDIM],
                  int n, real dt);
void put_snapshot(const real mass[], const real pos[][NDIM],
                  const real vel[][NDIM], int n, real t);
bool read_options(int argc, char *argv[], real & dt_param, real & dt_dia,
                  real & dt_out, real & dt_tot, bool & i_flag, bool & x_flag);
void write_diagnostics(const real mass[], const real pos[][NDIM],
                       const real vel[][NDIM], const real acc[][NDIM],
                       const real jerk[][NDIM], int n, real t, real epot,
                       int nsteps, real & einit, bool init_flag,
                       bool x_flag);
```

We start with `#include` statements to various libraries. The comments on each line mention some of the functions used from those libraries. If we would leave out one of these include statements, the corresponding functions listed could not be linked, and the compiler would issue an error.

The next statement indicates that we used the standard C++ namespace. Later, when gravitylab will have grown sufficiently large, it may be useful to create our own namespaces, in order to avoid collisions with other programs that may use names that are the same as we have chosen. Right now it is too early to worry about such complications.

The `typedef` statement defines the word `real` as an alternative for the build-in function type `double`. From now on we will only use the name `real` to indicate the standard floating point type `double`. It is far more logical to talk about real numbers of type `real`, together with the integers of type `int`, without using the archaic term 'double' that stems from the expression 'double precision' (long ago, the standard precision for floating point calculations used four bytes per floating point word, leading to the expression double precision for the now-standard eight-byte word length).

Next we introduce the symbol `NDIM` for the number of dimensions. So far we have simply used the number 3 in our loops over Cartesian coordinates, but it is much better not to have any magic numbers in a code, where a magic number is defined as anything that is not 0 or 1. The term "NDIM" for the number of dimensions is far clearer than a blind "3" in the middle of a piece of code. A second advantage of introducing a symbol, rather than magic numbers, is that we can change the symbol at one place, while guaranteeing its substitution everywhere else in the code. In the vast majority of cases, we will do our simulations in three spatial dimensions, hence the assignment here of the number 3 to `NDIM` here, but we will also encounter cases where we want to do some experimentation in one or two dimensions. In that case, changing 3 to 1 or 2 in this line is all we need to do (apart from making sure that we have not used uniquely three-dimensional constructs elsewhere in the code, such as for example the use of 3D spherical harmonics).

Note that older C-style usage would have defined `NDIM` through the macro definition "`#define NDIM 3`" . Nowadays, however, it is considered good form to use the C++ expression "`const int NDIM = 3;`" . Although the use of a `#define` macro in this case is quite innocent, there are many other cases where the use of macros can lead to code that is prone to confusing errors that are hard to debug. Therefore, as a matter of style it is a good idea to avoid them as much as possible.

The following nine function declarations are necessary if we want to have the freedom to define them in an arbitrary way in the rest of the file. The problem is that the C compiler goes through the file in one single pass, from top to bottom. As long as each function is invoked only after it has been seen by the compiler, there is no problem. In the codes `hermite4.C` through `hermite6.C`, the two functions listed at the top of the files were invoked only by `main()`, which was listed last. In general, however, with many functions there may not be a unique flow of functions calls. Besides, it is easier to follow the logic of the code if we can start with `main()` at the top of the file. The latter immediately implies that we will have to declare all functions mentioned in `main()`.

This need for redundant information in the form of declarations is a weakness of C++. In general, any time that a computer language forces you to duplicate information, it brings with it the danger of errors creeping in. It is easy to change the definition of a function without changing the declaration, or *vice versa.* In some cases, the compiler may catch this, but there may be other cases where overloading of function names with different argument sets makes it impossible for the compiler to catch such mistakes. Unfortunately, we will have to live with this situation.

Another example of redundant information in our program is the description of the command line options. Almost the same words appear once in the 'usage' part of the initial commments, and twice in the function `read_options()` (for the help option and the unknown option). It is possible to capture that information in a string at the top of the program, and to echo that string in `read_options()`. We will make such a modification later.

8.5 The Function `main()`

```
/*-----------------------------------------------------------------------------
 *  main  --  reads option values, reads a snapshot, and launches the
 *            integrator
 *-----------------------------------------------------------------------------
 */

int main(int argc, char *argv[])
{
    real  dt_param = 0.03;     // control parameter to determine time step size
    real  dt_dia = 1;          // time interval between diagnostics output
    real  dt_out = 1;          // time interval between output of snapshots
    real  dt_tot = 10;         // duration of the integration
    bool  init_out = false;    // if true: snapshot output with start at t = 0
                               //          with an echo of the input snapshot
    bool  x_flag = false;      // if true: extra debugging diagnostics output

    if (! read_options(argc, argv, dt_param, dt_dia, dt_out, dt_tot, init_out,
                       x_flag))
        return 1;                // halt criterion detected by read_options()

    int n;                       // N, number of particles in the N-body system
    cin >> n;

    real t;                      // time
    cin >> t;

    real * mass = new real[n];                  // masses for all particles
    real (* pos)[NDIM] = new real[n][NDIM];     // positions for all particles
    real (* vel)[NDIM] = new real[n][NDIM];     // velocities for all particles

    get_snapshot(mass, pos, vel, n);

    evolve(mass, pos, vel, n, t, dt_param, dt_dia, dt_out, dt_tot, init_out,
           x_flag);

    delete[] mass;
```

```
    delete[] pos;
    delete[] vel;
}
```

The first six variables declared at the top of `main()` receive their values from the function `read_options()` which reads the Unix style command line arguments. Note that each variable has a default value, which is retained unless it is changed explicitly by the corresponding command. We discuss the usage of command line options in the next section.

If the function `read_options()` detects a request for help, or the invocation of a non-existent option, it will return the Boolean value `false`. In that case the statement `!read_options()` is true, and program execution is halted. In C++, returning the value 0 indicates normal successful completion of the `main()` program, while any other value indicates a failure of some kind or other. For simplicity we return here the value 1.

Once the options are interpreted, we are ready to read the N-body snapshot from the standard input (which typically is redirected to read either the contents of a file, as in `nbody_sh1 < data.in` or to receive data from another program through a pipe, as in `generate_data | nbody_sh1`). Once the number of particles `n` has been read in, we can allocate storage space to contain the masses and dynamical information for all `n` particles, as we have seen in the previous chapter. The actual initialization of the arrays is carried out by the function `get_snapshot()`.

The real work is then delegated to the function `evolve()`, which oversees the evolution in time of the N-body system. When the call to `evolve()` returns, there is nothing left to be done. For good form we then deallocate the memory that we had dynamically allocated with the `new` operator. Note the square brackets in `delete`, which tell the compiler to delete the full memory assigned to the arrays. If we would leave this out, for example in a statement `delete[] mass`, we would only free the memory for `mass[0]`. This would constitute a memory leak, since the rest of the array will still be allocated, but it will be no longer usable in our program. In our particular case, this is no problem since we are about to terminate the program anyway, but in more complex cases, such as we will encounter in the function `evolve()`, it will be important to not create memory leaks.

8.6 Command Line Options

There are six command line options, Unix style, from which we can choose. All essential options have default values, so it is perfectly possible to run our code without specifying any of them. For example, if we start with an N-body snapshot in an input file `data.in`, we can run the code to produce a stream of snapshot data in the output file `data.out`, by typing:

```
|gravity> nbody_sh1 < data.in > data.out
```

This will have the exact same effect as if we would have specified the default values for the four main options, namely the time step control parameter (0.03), the interval between diagnostics output (1 time unit), the interval between output of snapshots (1 time unit), and the duration of the integration (10 time units):

```
|gravity> nbody_sh1 -d 0.03 -e 1 -o 1 -t 10 < data.in > data.out
```

If we would like to have three times smaller time steps, twice as many diagnostics outputs and with additional information, snapshot output intervals of 5 time units but starting at `t = 0`, and a total run time of 30 time units, we have to give the following command:

```
|gravity> nbody_sh1 -d 0.01 -e 0.5 -x -o 5 -i -t 30 < data.in > data.out
```

The order of the arguments is unimportant, but each option that expects a value (the `-d`, `-e`, `-o`, `-t` options) should be immediately followed by its corresponding value. By the way, the value `0.03` as the default for the scale of the time step parameter is somewhat arbitrary. In practice, a value of `0.1` is often found to be too large, while `0.01` is often overkill. For example, when we start from the initial conditions for three stars on a figure 8 orbit, running `nbody_sh1` with all default values in place, we wind up at time `t = 10` with a relative energy error of order 10^{-7}.

Of course, the optimal choice of values depend strongly on the particular application, and the default values are only a hint, in a blind attempt to come up with at least somewhat reasonable starting values. It is up to the user to make sure that these values are appropriate in a given situation, and if not, to supply a better value after some experimentation.

The help option can be invoked by typing:

```
|gravity> nbody_sh1 -h
```

This will not result in program execution, only in the printing of a short message that lays out the various command line option choices. A similar message will appear when we attempt to supply an non-existent option, for example:

```
|gravity> nbody_sh1 -q
nbody_sh1: invalid option -- q
usage: nbody_sh1 [-h (for help)] [-d step_size_control_parameter]
         [-e diagnostics_interval] [-o output_interval]
         [-t total_duration] [-i (start output at t = 0)]
         [-x (extra debugging diagnostics)]
|gravity>
```

All this behavior can be inspected in the function `read_options()`:

```
/*-----------------------------------------------------------------------------
 *  read_options  --  reads the command line options, and implements them.
 *
 *  note: when the help option -h is invoked, the return value is set to false,
 *        to prevent further execution of the main program; similarly, if an
 *        unknown option is used, the return value is set to false.
 *-----------------------------------------------------------------------------
 */

bool read_options(int argc, char *argv[], real & dt_param, real & dt_dia,
                  real & dt_out, real & dt_tot, bool & i_flag, bool & x_flag)
{
    int c;
    while ((c = getopt(argc, argv, "hd:e:o:t:ix")) != -1)
        switch(c){
            case 'h': cerr << "usage: " << argv[0]
                           << " [-h (for help)]"
                           << " [-d step_size_control_parameter]\n"
                           << "         [-e diagnostics_interval]"
                           << " [-o output_interval]\n"
                           << "         [-t total_duration]"
                           << " [-i (start output at t = 0)]\n"
                           << "         [-x (extra debugging diagnostics)]"
                           << endl;
                      return false;         // execution should stop after help
            case 'd': dt_param = atof(optarg);
                      break;
            case 'e': dt_dia = atof(optarg);
                      break;
            case 'i': i_flag = true;
                      break;
            case 'o': dt_out = atof(optarg);
                      break;
            case 't': dt_tot = atof(optarg);
                      break;
            case 'x': x_flag = true;
                      break;
            case '?': cerr << "usage: " << argv[0]
                           << " [-h (for help)]"
                           << " [-d step_size_control_parameter]\n"
                           << "         [-e diagnostics_interval]"
                           << " [-o output_interval]\n"
                           << "         [-t total_duration]"
                           << " [-i (start output at t = 0)]\n"
                           << "         [-x (extra debugging diagnostics)]"
```

```
                                  << endl;
                        return false;        // execution should stop after error
            }

    return true;                           // ready to continue program execution
}
```

Note that the six variables corresponding to the command line arguments are all passed by reference, so that the results are available to the calling program `main()`.

The function `getopt()` is a standard C library function that can be used equally well in C++ programs. Its third argument is a string which lists all command line options. Each option can only consist of a single letter. Those letters that should be followed by a value to be read in are indicated by a colon immediately following the letter. The string `"hd:e:o:t:ix"` tells us that options `h`, `i` and `x` do not expect additional values, while options `d`, `e`, `o` and `t` are to be followed with an argument, all of which are of type `real` in our particle case. All option arguments are by default passed as ASCII strings, so we need the function `atof()` to convert the ASCII information into the proper floating point value, as we already saw in the previous chapter.

Notice that each `case` in the body of the `switch` statement is ended by either a `return` statement or a `break` statement. The latter is necessary, since the default behavior of `switch` is to 'fall through' from one case to the next, something that is clearly not desirable here. After we jump out of the `switch` statement through a `break` command, we encounter the last statement, "`return true;`" which tells the calling program that all is well, and that execution can continue.

8.7 Snapshot Input

The code for snapshot input is straightforward:

```
/*-----------------------------------------------------------------------------
 *  get_snapshot  --  reads a single snapshot from the input stream cin.
 *
 *  note: in this implementation, only the particle data are read in, and it
 *        is left to the main program to first read particle number and time
 *-----------------------------------------------------------------------------
 */

void get_snapshot(real mass[], real pos[][NDIM], real vel[][NDIM], int n)
{
    for (int i = 0; i < n ; i++){
        cin >> mass[i];                       // mass of particle i
        for (int k = 0; k < NDIM; k++)
```

```
            cin >> pos[i][k];                  // position of particle i
        for (int k = 0; k < NDIM; k++)
            cin >> vel[i][k];                  // velocity of particle i
    }
}
```

Note that we do not check here whether a complete snapshot is being offered on the standard input stream in the right format. It would be better to verify, for example, that new lines
`n` occur in the correct places, separating each particle, and that no end-of-file condition is encountered before the whole N-body snapshot is read in. In later versions we will provide more complete error checking.

8.8 Snapshot Output

The code for snapshot output is similarly simple:

```
/*-----------------------------------------------------------------------------
 *  put_snapshot  --  writes a single snapshot on the output stream cout.
 *
 *  note: unlike get_snapshot(), put_snapshot handles particle number and time
 *-----------------------------------------------------------------------------
 */

void put_snapshot(const real mass[], const real pos[][NDIM],
                  const real vel[][NDIM], int n, real t)
{
    cout.precision(16);                        // full double precision

    cout << n << endl;                         // N, total particle number
    cout << t << endl;                         // current time
    for (int i = 0; i < n ; i++){
        cout << mass[i];                       // mass of particle i
        for (int k = 0; k < NDIM; k++)
            cout << ' ' << pos[i][k];          // position of particle i
        for (int k = 0; k < NDIM; k++)
            cout << ' ' << vel[i][k];          // velocity of particle i
        cout << endl;
    }
}
```

Note that the masses, positions, and velocities are all declared as `const` in the declaration of the function arguments. This means that this function is not allowed to change the values of those particular arguments. Being able to specify function arguments as `const` is a very useful C++ feature. It can help the compiler by providing extra information; it allows the compiler to flag an error if in the body of the function an attempt is made to change one of those arguments erroneously; and most importantly, it gives the human reader useful information about the intentions of the programmer.

For all these reasons, it is important to be consistent in the use of `const` specifications, and to always use `const` wherever we can. When we do this, we thereby imply that the absence of a `const` specifier for an argument means that we do want to affect the value of that particular argument. For example, in the previous function `get_snapshot()`, masses, positions, and velocities are not preceded by `const`. Indeed, all three arrays are being initialized in that function, and it is useful to be able to anticipate that already from looking at the argument list, either here or at the top of the file where all functions are declared.

The first line of the body of the function sets the precision for all subsequent output. It turns out that eight-byte double precision corresponds to about 16 digits of relative accuracy. If we would output less than 16 significant digits for each `real` variable, we would lose information. A subsequent program reading in the snapshot that we have just written out would not have access to the full information that we had before we wrote our data. On the other hand, if we would output those numbers with more than 16 digits, the extra digits would be effective garbage. While this doesn't hurt, it is a waste of space (and possibly later processing time) to go beyond 16 digits.

8.9 Reporting Diagnostics

Here is the code for the function that writes diagnostics to the standard error stream. Note the declarations of arguments: all arrays are specified to be `const`, which is appropriate since their values should only be reported, without changing them. The argument `einit` is passed by reference, since it will hold the initial value of the total energy of the system, information that should be passed back to the calling function. The other arguments are all passed by value.

```
/*-----------------------------------------------------------------------------
 *  write_diagnostics  --  writes diagnostics on the error stream cerr:
 *                         current time; number of integration steps so far;
 *                         kinetic, potential, and total energy; absolute and
 *                         relative energy errors since the start of the run.
 *                         If x_flag (x for eXtra data) is true, all internal
 *                         data are dumped for each particle (mass, position,
 *                         velocity, acceleration, and jerk).
 *
 *  note: the kinetic energy is calculated here, while the potential energy is
 *        calculated in the function get_acc_jerk_pot_coll().
```

```
 *-----------------------------------------------------------------------------
 */

void write_diagnostics(const real mass[], const real pos[][NDIM],
                       const real vel[][NDIM], const real acc[][NDIM],
                       const real jerk[][NDIM], int n, real t, real epot,
                       int nsteps, real & einit, bool init_flag,
                       bool x_flag)
{
    real ekin = 0;                        // kinetic energy of the n-body system
    for (int i = 0; i < n ; i++)
        for (int k = 0; k < NDIM ; k++)
            ekin += 0.5 * mass[i] * vel[i][k] * vel[i][k];

    real etot = ekin + epot;              // total energy of the n-body system

    if (init_flag)                        // at first pass, pass the initial
        einit = etot;                     // energy back to the calling function

    cerr << "at time t = " << t << " , after " << nsteps
         << " steps :\n  E_kin = " << ekin
         << " , E_pot = " << epot
         << " , E_tot = " << etot << endl;
    cerr << "                "
         << "absolute energy error: E_tot - E_init = "
         << etot - einit << endl;
    cerr << "                "
         << "relative energy error: (E_tot - E_init) / E_init = "
         << (etot - einit) / einit << endl;

    if (x_flag){
        cerr << "  for debugging purposes, here is the internal data "
             << "representation:\n";
        for (int i = 0; i < n ; i++){
            cerr << "    internal data for particle " << i+1 << " : " << endl;
            cerr << "      ";
            cerr << mass[i];
            for (int k = 0; k < NDIM; k++)
                cerr << ' ' << pos[i][k];
            for (int k = 0; k < NDIM; k++)
                cerr << ' ' << vel[i][k];
            for (int k = 0; k < NDIM; k++)
                cerr << ' ' << acc[i][k];
            for (int k = 0; k < NDIM; k++)
                cerr << ' ' << jerk[i][k];
            cerr << endl;
        }
```

```
    }
}
```

The only calculation performed in this function is that of the kinetic energy. The potential energy is determined in the function `get_acc_jerk_pot_coll()`. The `init_flag` is set to `true` when `write_diagnostics()` is evoked for the first time, at `t = 0`. In that case, we want to pass the value of the initial total energy back to the calling function `evolve()`, which can use that information to compare it with later measured values of the total energy, in order to determine the absolute and relative amounts of energy drifts, which are a good measure of numerical accuracy.

Note that we could have defined the initial energy `einit` as a static variable inside `write_diagnostics()`. For our present purpose that would be fine, but this type of programming may easily create a future limitation. If some day we would like to compare two different N-body systems, each of which evolves, we would get into a conflict if both of them would try to access the same static variable. Therefore, for the same reason we don't use global variables in the first place, we prefer to pass `einit` as a function variable.

8.10 Orbit Integration

We now come to the function that manages the orbit evolution, driving the Hermite integrator and scheduling the various output operations:

```
/*-----------------------------------------------------------------------------
 *  evolve  --  integrates an N-body system, for a total duration dt_tot.
 *              Snapshots are sent to the standard output stream once every
 *              time interval dt_out.  Diagnostics are sent to the standard
 *              error stream once every time interval dt_dia.
 *
 *  note: the integration time step, shared by all particles at any given time,
 *        is variable.  Before each integration step we use coll_time (short
 *        for collision time, an estimate of the time scale for any significant
 *        change in configuration to happen), multiplying it by dt_param (the
 *        accuracy parameter governing the size of dt in units of coll_time),
 *        to obtain the new time step size.
 *
 *  Before moving any particles, we start with an initial diagnostics output
 *  and snapshot output if desired.  In order to write the diagnostics, we
 *  first have to calculate the potential energy, with get_acc_jerk_pot_coll().
 *  That function also calculates accelerations, jerks, and an estimate for the
 *  collision time scale, all of which are needed before we can enter the main
 *  integration loop below.
```

```
 *      In the main loop, we take as many integration time steps as needed to
 *  reach the next output time, do the output required, and continue taking
 *  integration steps and invoking output this way until the final time is
 *  reached, which triggers a `break' to jump out of the infinite loop set up
 *  with `while(true)'.
 *-----------------------------------------------------------------------------
 */

void evolve(const real mass[], real pos[][NDIM], real vel[][NDIM],
            int n, real & t, real dt_param, real dt_dia, real dt_out,
            real dt_tot, bool init_out, bool x_flag)
{
    cerr << "Starting a Hermite integration for a " << n
         << "-body system,\n  from time t = " << t
         << " with time step control parameter dt_param = " << dt_param
         << "  until time " << t + dt_tot
         << " ,\n  with diagnostics output interval dt_dia = "
         << dt_dia << ",\n  and snapshot output interval dt_out = "
         << dt_out << "." << endl;

    real (* acc)[NDIM] = new real[n][NDIM];          // accelerations and jerks
    real (* jerk)[NDIM] = new real[n][NDIM];         // for all particles
    real epot;                         // potential energy of the n-body system
    real coll_time;                    // collision (close encounter) time scale

    get_acc_jerk_pot_coll(mass, pos, vel, acc, jerk, n, epot, coll_time);

    int nsteps = 0;                // number of integration time steps completed
    real einit;                    // initial total energy of the system

    write_diagnostics(mass, pos, vel, acc, jerk, n, t, epot, nsteps, einit,
                      true, x_flag);
    if (init_out)                                    // flag for initial output
        put_snapshot(mass, pos, vel, n, t);

    real t_dia = t + dt_dia;           // next time for diagnostics output
    real t_out = t + dt_out;           // next time for snapshot output
    real t_end = t + dt_tot;           // final time, to finish the integration

    while (true){
        while (t < t_dia && t < t_out && t < t_end){
            real dt = dt_param * coll_time;
            evolve_step(mass, pos, vel, acc, jerk, n, t, dt, epot, coll_time);
            nsteps++;
        }
        if (t >= t_dia){
            write_diagnostics(mass, pos, vel, acc, jerk, n, t, epot, nsteps,
```

```
                            einit, false, x_flag);
            t_dia += dt_dia;
        }
        if (t >= t_out){
            put_snapshot(mass, pos, vel, n, t);
            t_out += dt_out;
        }
        if (t >= t_end)
            break;
    }

    delete[] acc;
    delete[] jerk;
}
```

Starting again with the argument list, we see that the mass array, as always, is defined as `const`, since we do not model a mechanism for mass loss for stars, nor do we (yet) allow collisions between stars, which could be followed by mergers that would produce a merger remnant with a mass equal to the sum of the masses of the two stars. The only place where we do not define the mass array as `const` is in the function `get_snapshot`, where the mass values are read in from the standard input stream. Note that the time `t` is passed by reference. In our current program, this is not necessary, since the value of `t` is not used in `main()`, where execution is halted immediately upon completion of the call to `evolve()`. However, in future extensions we may well add further commands in `main()`, and in that case it would be useful to have the value of the current time available.

As we have seen before, before we can enter the integration loop we have to start with an initial call to the function computing the accelerations and jerks. This function, `get_acc_jerk_pot_coll()` does what its name suggest: besides calculating accelerations and jerks, it also reports the value of the total potential energy of the system as well as the value of the time scale on which a 'collision' between particles can occur, *i. e.* a significant change of order unity in the local configuration of at least two particles. The latter information, stored in the variable `coll_time`, will be needed in the main integration loop in order to determine the size of the first time step. Accelerations and jerks are needed for the first part of the first integration time step, and the potential energy is used in the initial call to `write_diagnostics()`, following the first call to `get_acc_jerk_pot_coll()`.

In addition, if the user has specified the `init_out` flag to be true, the input values of the N-body system are echoed as they are on the output stream; the default behavior is to wait with output until some integration steps have been taken. This is a sensible default, since in many cases we are only interested in one final output snapshot, which can then served as the input for a later invocation of the integrator. If we invoke our program with the same value for the snapshot output interval as the duration of the run, we guarantee that only one final output will be made. An example usage of this type is:

```
|gravity> nbody_sh1 -d 0.01 -e 2 -o 40 -t 40 < data.in > data.out
```

Before entering the main integration loop, we schedule the next times for diagnostics and snapshot output, as well as the final halting time. The loop itself is an infinite loop, governed by the tautological `while (true)`, which is obviously always the case. The standard C/C++ trick to define an infinite loop uses an empty for loop, in the form `for(;;)`, but that expression is less transparent, whereas `while (true)` leaves no doubt as to it being an infinite loop. The only way to jump out of this infinite loop is at the end of the loop: when time progresses past the halting time `t_end`, the `break` statement causes control flow to continue past the loop.

The first time we enter the loop, the second `while` argument will be evaluated as `true`, unless one of the three values `dt_dia, dt_out` or `dt_tot` would be zero or negative, which would be nonsensical values. Ideally, we should check somewhere that all command line option arguments fall within reasonable ranges. Since in the present code we have already introduced so many new features, we will not include such a defensive programming style at this point. However, later on we will insist on checking all values which reach a program through an interface, such as presented by command line options. For now, we will live with the danger of a non-positive value for either `dt_dia` or `dt_out`, which combined with a positive value for `dt_tot` would lead to an infinite number of output attempts, without the time ever advancing.

With natural choices of parameters, the majority of loop cycles will not lead to any output. In those cases a new time step size is determined, and the function `evolve_step()` is called, which as the name implies will advance the system by one integration step, and in addition update the time by an amount `dt`. Sooner or later it will be time for output or for ending the run. In either case, the second `while` statement will evaluate as `false`, no integration time step will be taken and therefore the time will not be advanced either. Instead, the required output will be done and/or the integration will be finished altogether. If the run is not yet finished, the next cycle in the infinite loop will lead to another integration step, and so on.

Note the freeing up of memory for acceleration and jerk arrays, at the end of `evolve()`. As in the case of the memory allocation in `main()`, this is not strictly necessary, since the program is about to finish, but again it is certainly good form to include these statements here.

8.11 Taking a Single Integration Step

In the function `evolve_step()`, we encounter the first case where specific memory allocation and deallocation occurs more often than once during a run:

```
/*-----------------------------------------------------------------------------
 *  evolve_step  --  takes one integration step for an N-body system, using the
 *                   Hermite algorithm.
 *-----------------------------------------------------------------------------
 */

void evolve_step(const real mass[], real pos[][NDIM], real vel[][NDIM],
                 real acc[][NDIM], real jerk[][NDIM], int n, real & t,
                 real dt, real & epot, real & coll_time)
{
    real (* old_pos)[NDIM] = new real[n][NDIM];
    real (* old_vel)[NDIM] = new real[n][NDIM];
    real (* old_acc)[NDIM] = new real[n][NDIM];
    real (* old_jerk)[NDIM] = new real[n][NDIM];

    for (int i = 0; i < n ; i++)
        for (int k = 0; k < NDIM ; k++){
          old_pos[i][k] = pos[i][k];
          old_vel[i][k] = vel[i][k];
          old_acc[i][k] = acc[i][k];
          old_jerk[i][k] = jerk[i][k];
        }

    predict_step(pos, vel, acc, jerk, n, dt);
    get_acc_jerk_pot_coll(mass, pos, vel, acc, jerk, n, epot, coll_time);
    correct_step(pos, vel, acc, jerk, old_pos, old_vel, old_acc, old_jerk,
                 n, dt);
    t += dt;

    delete[] old_pos;
    delete[] old_vel;
    delete[] old_acc;
    delete[] old_jerk;
}
```

As we have seen already in chapter 6, the Hermite code requires knowledge of the values of all four dynamical variables at the previous time step, indicated here by the prefix `old_`. Since we do not want to introduce global variables, and since these variables are not needed outside the context of the current function, we allocate the memory in the first four lines, and free up those memory locations in the last four lines. If we now would omit those last four lines, the resulting memory leak could let us run into serious trouble. For example, taking a million time steps with a hundred-body system would cause us to loose $4 * NDIM = 12$ words or $12 * 8 = 96$ bytes for each particle for each time step, leading to a total memory loss of $96 * 10^2 * 10^6$ bytes or roughly ten Gbytes, which may well be larger than the core memory of the computer at hand.

Again, it would be very good if we would check with each call to `new` whether there is still enough memory available. Since we do not do that here, a memory leak will suddenly cause the program to crash, without giving us any clue of where to look. Even using a debugger may not help, since the actual crash may well occur somewhere else, where a small amount of legitimate memory is requested, only to find out that all memory has just been exhausted elsewhere in the code. Once more, we will postpone but not neglect this type of defensive programming.

After the current values of the dynamical variables have been passed to the `old_` copies, we take the first half of a Hermite pass, in a call to `predict_step()`, followed by a recalculation of accelerations and jerks, as well as potential energy and collision time scale. We are then ready to complete the Hermite step through a call to `correct_step()`, and update the time `t`.

8.12 The Predictor Step

The first half of a Hermite step is particularly simple, nothing more than a rather short Taylor series development:

```
/*-----------------------------------------------------------------------------
 *  predict_step  --  takes the first approximation of one Hermite integration
 *                    step, advancing the positions and velocities through a
 *                    Taylor series development up to the order of the jerks.
 *-----------------------------------------------------------------------------
 */

void predict_step(real pos[][NDIM], real vel[][NDIM],
                  const real acc[][NDIM], const real jerk[][NDIM],
                  int n, real dt)
{
    for (int i = 0; i < n ; i++)
        for (int k = 0; k < NDIM ; k++){
            pos[i][k] += vel[i][k]*dt + acc[i][k]*dt*dt/2
                                      + jerk[i][k]*dt*dt*dt/6;
            vel[i][k] += acc[i][k]*dt + jerk[i][k]*dt*dt/2;
        }
}
```

Notice how much we can already read off from the way the arguments to `predict_step()` are declared: accelerations and jerks are passed as `const` variables, whereas positions and velocities are not. This implies that the latter two are updated, whereas the former two are used to provide information for the update, without being changed themselves. This of course is exactly what happens.

8.13 The Corrector Step

The second half of a Hermite step is again a Taylor series development, this time to a higher order than in the predictor step, even though this is not obvious from the way it is written. We refer to the discussion in the beginning of chapter 6, where the Taylor series character of the corrector step is made explicit. Here is the code:

```
/*-----------------------------------------------------------------------------
 *  correct_step  --  takes one iteration to improve the new values of position
 *                    and velocities, effectively by using a higher-order
 *                    Taylor series constructed from the terms up to jerk at
 *                    the beginning and the end of the time step.
 *-----------------------------------------------------------------------------
 */

void correct_step(real pos[][NDIM], real vel[][NDIM],
                  const real acc[][NDIM], const real jerk[][NDIM],
                  const real old_pos[][NDIM], const real old_vel[][NDIM],
                  const real old_acc[][NDIM], const real old_jerk[][NDIM],
                  int n, real dt)
{
    for (int i = 0; i < n ; i++)
        for (int k = 0; k < NDIM ; k++){
            vel[i][k] = old_vel[i][k] + (old_acc[i][k] + acc[i][k])*dt/2
                                      + (old_jerk[i][k] - jerk[i][k])*dt*dt/12;
            pos[i][k] = old_pos[i][k] + (old_vel[i][k] + vel[i][k])*dt/2
                                      + (old_acc[i][k] - acc[i][k])*dt*dt/12;
        }
}
```

8.14 Where All the Work is Done

We now arrive at the core function of `nbody_sh1.C`, where all the hard work is being done. In addition, this function is both the longest and the most complicated among the ten functions in the file. The main reason for the complexity is that we are trying to accomplish four things in one function, as the name indicates. While calculating accelerations and jerks are logically related, the calculation of the potential energy and the collision time is more a matter of convenience with little natural or logical relation to the calculation of the first two. The main reason for bundling these four operations is efficiency. Here is the code:

```
/*-----------------------------------------------------------------------------
 *  get_acc_jerk_pot_coll  --  calculates accelerations and jerks, and as side
 *                             effects also calculates potential energy and
 *                             the time scale coll_time for significant changes
 *                             in local configurations to occur.
 *                                                  --                     --
 *                                                 |          -->  -->       |
 *               M                           M     |           r  . v        |
 *   -->          j    -->       -->          j    | -->        ji   ji -->  |
 *    a   == -------- r    ;    j   ==  -------- |  v   - 3 ---------  r   |
 *     ji     |-->  |3   ji        ji     |-->  |3 |   ji     |-->  |2    ji |
 *            | r   |                     | r   |  |          | r   |        |
 *            |  ji |                     |  ji |  |__        |  ji |      __|
 *
 *  note: it would be cleaner to calculate potential energy and collision time
 *        in a separate function.  However, the current function is by far the
 *        most time consuming part of the whole program, with a double loop
 *        over all particles that is executed every time step.  Splitting off
 *        some of the work to another function would significantly increase
 *        the total computer time (by an amount close to a factor two).
 *
 *  We determine the values of all four quantities of interest by walking
 *  through the system in a double {i,j} loop.  The first three, acceleration,
 *  jerk, and potential energy, are calculated by adding successive terms;
 *  the last, the estimate for the collision time, is found by determining the
 *  minimum value over all particle pairs and over the two choices of collision
 *  time, position/velocity and sqrt(position/acceleration), where position and
 *  velocity indicate their relative values between the two particles, while
 *  acceleration indicates their pairwise acceleration.  At the start, the
 *  first three quantities are set to zero, to prepare for accumulation, while
 *  the last one is set to a very large number, to prepare for minimization.
 *       The integration loops only over half of the pairs, with j > i, since
 *  the contributions to the acceleration and jerk of particle j on particle i
 *  is the same as those of particle i on particle j, apart from a minus sign
 *  and a different mass factor.
 *-----------------------------------------------------------------------------
 */

void get_acc_jerk_pot_coll(const real mass[], const real pos[][NDIM],
                           const real vel[][NDIM], real acc[][NDIM],
                           real jerk[][NDIM], int n, real & epot,
                           real & coll_time)
{
    for (int i = 0; i < n ; i++)
```

```
        for (int k = 0; k < NDIM ; k++)
            acc[i][k] = jerk[i][k] = 0;
    epot = 0;
    const real VERY_LARGE_NUMBER = 1e300;
    real coll_time_q = VERY_LARGE_NUMBER;      // collision time to 4th power
    real coll_est_q;                           // collision time scale estimate
                                               // to 4th power (quartic)
    for (int i = 0; i < n ; i++){
        for (int j = i+1; j < n ; j++){            // rji[] is the vector from
            real rji[NDIM];                        // particle i to particle j
            real vji[NDIM];                        // vji[] = d rji[] / d t
            for (int k = 0; k < NDIM ; k++){
                rji[k] = pos[j][k] - pos[i][k];
                vji[k] = vel[j][k] - vel[i][k];
            }
            real r2 = 0;                           // | rji |^2
            real v2 = 0;                           // | vji |^2
            real rv_r2 = 0;                        // ( rij . vij ) / | rji |^2
            for (int k = 0; k < NDIM ; k++){
                r2 += rji[k] * rji[k];
                v2 += vji[k] * vji[k];
                rv_r2 += rji[k] * vji[k];
            }
            rv_r2 /= r2;
            real r = sqrt(r2);                     // | rji |
            real r3 = r * r2;                      // | rji |^3

// add the {i,j} contribution to the total potential energy for the system:

            epot -= mass[i] * mass[j] / r;

// add the {j (i)} contribution to the {i (j)} values of acceleration and jerk:

            real da[3];                            // main terms in pairwise
            real dj[3];                            // acceleration and jerk
            for (int k = 0; k < NDIM ; k++){
                da[k] = rji[k] / r3;                           // see equations
                dj[k] = (vji[k] - 3 * rv_r2 * rji[k]) / r3;    // in the header
            }
            for (int k = 0; k < NDIM ; k++){
                acc[i][k] += mass[j] * da[k];                 // using symmetry
                acc[j][k] -= mass[i] * da[k];                 // find pairwise
                jerk[i][k] += mass[j] * dj[k];                // acceleration
                jerk[j][k] -= mass[i] * dj[k];                // and jerk
            }

// first collision time estimate, based on unaccelerated linear motion:
```

```
            coll_est_q = (r2*r2) / (v2*v2);
            if (coll_time_q > coll_est_q)
                coll_time_q = coll_est_q;

// second collision time estimate, based on free fall:

            real da2 = 0;                                  // da2 becomes the
            for (int k = 0; k < NDIM ; k++)                // square of the
                da2 += da[k] * da[k];                      // pair-wise accel-
            double mij = mass[i] + mass[j];                // eration between
            da2 *= mij * mij;                              // particles i and j

            coll_est_q = r2/da2;
            if (coll_time_q > coll_est_q)
                coll_time_q = coll_est_q;
        }
    }                                              // from q for quartic back
    coll_time = sqrt(sqrt(coll_time_q));           // to linear collision time
}
```

Notice the distribution of `const` declarations here, which is just the opposite from what we saw in `predict_step()` and `correct_step()`. In the latter two accelerations and jerk were `const` while positions and velocities were updated. Here the roles are reversed. In addition, there are two variables that are called by reference, `epot` and `coll_time`, which enable the information about potential energy and collision time to flow back to the calling function `evolve_step()` and from there back to `evolve()`, where they are used, as we have seen above.

After preparing the proper initial values for the four variables of interest, we enter the $\{i, j\}$ loop running over all particle pairs. As we have seen in the previous two chapters, we first compute a number of auxiliary quantities before we are ready to calculate first the contribution of a pair of particles to the potential energy and then their mutual contributions to each others acceleration and jerk.

At the end of the loop, we compute the two different collision time step estimates, in the same way we discovered at the end of the previous chapter. The first estimate follows the approximate of unperturbed linear motion, extrapolating current separation and rate of change of separation in order to guess when the particles will change their relative configuration substantially. The second estimate neglects the current rate of change of the pairwise separation, estimating instead the free-fall time of the two particles, in case they would start off at rest. In practice, the smaller of the two estimates provides a reasonably safe estimate for the time scale on which significant changes in configuration can occur.

8.15 Closing Logo

At the very end of our file, we add a simpler version of the gravitylab logo that we encountered at the top of the file:

```
/*-----------------------------------------------------------------------------
 *                                                                  \\   o
 *  end of file:  nbody_sh1.C                                       /\\'  O
 *                                                                 /\     |
 *=============================================================================
 */
```

It contains the name of the file, for consistency, and it guarantees that no part of the file has been truncated in a process of copying, editing or transmission over the net. While such mishaps are very rare nowadays, they still can occur occasionally, and it seems prudent to mark the intended end of the file. Meanwhile, our intrepid observer has changed directions from which to observe the world.